Process Algebra: An Algebraic Theory of Concurrency

Wan Fokkink

Vrije Universiteit Amsterdam, Department of Theoretical Computer Science,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
wanf@cs.vu.nl

Abstract. This tutorial provides an overview of the process algebra ACP.

1 Introduction

The term "process algebra" was coined in 1982 by Jan Bergstra and Jan Willem Klop, originally in the sense of universal algebra, to refer to a structure satisfying a particular set of axioms. Nowadays it is used in a more general sense for algebraic approaches to describe and study concurrent processes. In the late 70's, Robin Milner and Tony Hoare largely independently developed the process algebras CSS and CSP, respectively. In the early 80's, Bergstra and Klop developed a third process algebra called ACP.

System behaviour generally consists of processes and data. Processes are the control mechanisms for the manipulation of data. While processes are dynamic and active, data is static and passive. System behaviour tends to be composed of several processes that are executed concurrently, where these processes exchange data in order to influence each other's behaviour. Fundamental to process algebra is a parallel operator, to break down systems into their concurrent components. A set of equations is imposed to derive whether two terms are behaviourally equivalent. In this framework, non-trivial properties of systems can be established in a rigorous and elegant fashion. For example, it may be possible to equate an implementation of a system to the specification of its required input/output relation. A variety of automated tools have been developed to facilitate the derivation of such properties in a process algebraic framework.

Abstract data types (see, e.g., [5]) offer a framework in which also the data can be specified by means of equations. μCRL [10] is a specification language, supported by verification tools, that combines process algebra with equational specification of data types. In this tutorial we will however mainly focus on processes.

Applications of process algebra exist in diverse fields such as safety-critical systems, network protocols, and biology. In the educational vein, process algebra has been recognised to teach skills to deal with complex concurrent systems, by representing and reasoning about such systems in a mathematically clear and precise manner.

Recommended textbooks are [15] for CCS, [17] for CSP, and [8] for ACP. Jos Baeten [2] presented a detailed account on the history of process algebra. Here I will focus on the process algebra ACP; this tutorial is based on [8].

This tutorial is structured as follows. Section 2 explains the general framework. Section 3 introduces the basic process algebra BPA. Section 4 extends the framework with parallelism, communication and encapsulation. Section 5 adds recursion to express infinite behaviour. Section 6 introduces the silent step τ, and abstraction operators to hide internal behaviour. Finally, Section 7 presents a process algebraic specification and verification of the Alternating Bit Protocol.

2 The General Framework

Process graphs As starting point, we assume that system behaviour is represented as a process graph. It basically consists of a set of nodes together with a set of labelled edges between these nodes. A node represents a system state, while a labelled edge represents a transition from one system state to the next. That is, if the process graph contains an edge $s \xrightarrow{a} s'$, then the process graph can evolve from state s into state s' by the execution of action a. One state is selected to be the root state, i.e., the initial state of the process.

Behavioural equivalences. The states in process graphs are distinguished by some behavioural equivalence. For example, such an equivalence may relate two process graphs if and only if their root states can execute exactly the same strings of actions. This tutorial focuses on bisimilarity, which is the finest of all known process equivalences. Bisimilarity requires not only that two process graphs can execute the same strings of actions, but also that they have the same branching structure. Bisimilarity is widely recognised as a well-suited semantic notion when reasoning about concurrent processes.

Process algebra terms. For the purpose of mathematical reasoning it is often convenient to represent process graphs algebraically in the form of terms. Process algebra focuses on the specification and manipulation of process terms as induced by a collection of operator symbols. This symbolic notation facilitates manipulation by a computer. Most process algebras contain basic operators to build finite processes, communication operators to express concurrency, and some notion of recursion to capture infinite behaviour. Moreover, it is convenient to introduce two special constants: the deadlock enables us to force actions into communication, while the silent step allows us to abstract away from internal computations.

Structural operational semantics. Transition rules, which are inductive proof rules, provide each process term with its intended process graph (see, e.g., [1]). We are going to present the process algebra ACP$_\tau$ with recursion in several steps, starting from the basic process algebra BPA. With every extension we need to check that it is conservative, meaning that the transition rules for the new operators do not influence the behaviour of the "old" process algebra terms. This can be checked by inspecting the syntactic form of the transition rules. Moreover, the process algebraic operators should be a congruence with respect to bisimilarity, meaning that if two process terms are bisimilar, then they are also bisimilar under any context. Again this can be checked by inspecting the syntactic form of the transition rules.

Equational logic. The crux of process algebra is that it imposes an equational logic on process terms that is sound and complete. Soundness means that if two process terms can be equated then their process graphs are behaviourally equivalent. Vice versa, completeness means that if two process terms have behaviourally equivalent process graphs, then they can be equated.

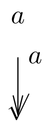
3 Basic Process Algebra

We start with describing a basic process algebra, denoted by BPA.

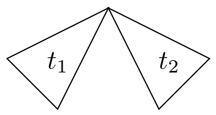
3.1 Syntax of BPA

The core for process algebra consists of the following operators.

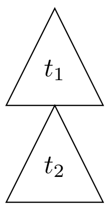
- First of all, we assume a non-empty set A of (atomic) actions, representing indivisible behaviour (such as reading a datum, or sending a datum). Each atomic action a is a constant that can execute itself, after which it terminates successfully:



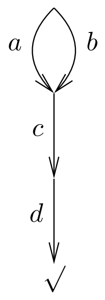
  The predicate $\xrightarrow{a} \surd$ represents successful termination after the execution of action a.
- Moreover, we assume a binary operator $+$ called alternative composition. The term $t_1 + t_2$ represents the process that executes the behaviour of either t_1 or t_2. In other words, the process graph of $t_1 + t_2$ is obtained by joining the process graphs of t_1 and t_2 at their root states:



- Finally, we assume a binary operator $\cdot$ called sequential composition. The term $t_1 \cdot t_2$ represents the process that executes first the behaviour of t_1, and then the behaviour of t_2. In other words, the process graph of $t_1 \cdot t_2$ is obtained by replacing each successful termination $s \xrightarrow{a} \surd$ in in the process graph of t_1 by a transition $s \xrightarrow{a} s'$, where s' is the root of the process graph of t_2:



Example 1. Let a, b, c and d be actions. The basic process term $((a+b)\cdot c)\cdot d$ represents the following process graph, with the root state presented at the top:



Each finite process graph can be represented by a process term that is built from the set A of atomic actions, $+$, and $\cdot$. Such terms are called basic process terms, and the collection of all basic process terms is called basic process algebra, abbreviated to BPA.

3.2 Transition Rules of BPA

We have provided a syntax for basic process terms, together with some intuition for the process graph that belongs to such a term. This relationship has to be made formal in order for it to become really meaningful. For this purpose we apply structural operational semantics. This involves giving a collection of transition rules, which define transitions $t \xrightarrow{a} t'$ to express that term t can evolve into term t' by the execution of action a, and predicates $t \xrightarrow{a} \surd$ to express that term t can terminate successfully by the execution of action a.

Table 1 presents the transition rules that constitute the structural operational semantics of BPA. The variables x, x', y and y' in the transition rules range over the collection of basic process terms, while v ranges over the set A of atomic actions.

The transition rules of BPA provide each basic process term with a process graph, according to the intuition that was presented in the previous section:

- the first transition rule says that each atomic action v can terminate successfully by executing itself;
- the next four transition rules express that $t+t'$ behaves as either t or t';

Table 1. Transition rules of BPA

$$\frac{}{v \xrightarrow{v} \surd}$$

$$\frac{x \xrightarrow{v} \surd}{x+y \xrightarrow{v} \surd} \qquad \frac{x \xrightarrow{v} x'}{x+y \xrightarrow{v} x'} \qquad \frac{y \xrightarrow{v} \surd}{x+y \xrightarrow{v} \surd} \qquad \frac{y \xrightarrow{v} y'}{x+y \xrightarrow{v} y'}$$

$$\frac{x \xrightarrow{v} \surd}{x\cdot y \xrightarrow{v} y} \qquad \frac{x \xrightarrow{v} x'}{x\cdot y \xrightarrow{v} x'\cdot y}$$

– the last two transition rules express that $t{\cdot}t'$ executes t until successful termination, after which it proceeds to execute t'.

Example 2. The transition rules in Table 1 provide the basic process term $((a+b){\cdot}c){\cdot}d$ with the following process graph (cf. Example 1):

$$\begin{tikzcd}
((a+b)\cdot c)\cdot d \arrow[d, bend right, "a"'] \arrow[d, bend left, "b"] \\
c\cdot d \arrow[d, "c"'] \\
d \arrow[d, "d"'] \\
\surd
\end{tikzcd}$$

For instance, the transition $((a+b){\cdot}c){\cdot}d \xrightarrow{b} c{\cdot}d$ can be proved from the transition rules in Table 1 as follows:

$$\begin{array}{lll}
b \xrightarrow{b} \surd & \Big(\dfrac{}{v \xrightarrow{v} \surd}, & v := b\Big) \\
\cline{1-1}
a+b \xrightarrow{b} \surd & \Big(\dfrac{y \xrightarrow{v} \surd}{x+y \xrightarrow{v} \surd}, & v := b,\ x := a,\ y := b\Big) \\
\cline{1-1}
(a+b){\cdot}c \xrightarrow{b} c & \Big(\dfrac{x \xrightarrow{v} \surd}{x{\cdot}y \xrightarrow{v} y}, & v := b,\ x := a+b,\ y := c\Big) \\
\cline{1-1}
((a+b){\cdot}c){\cdot}d \xrightarrow{b} c{\cdot}d & \Big(\dfrac{x \xrightarrow{v} x'}{x{\cdot}y \xrightarrow{v} x'{\cdot}y}, & v := b,\ x := (a+b){\cdot}c,\ x' := c,\ y := d\Big)
\end{array}$$

At the right-hand side, the transition rules are displayed that are applied in the consecutive proof steps, together with the substitutions that are applied to them.

From now on, as binding convention we assume that $\cdot$ binds stronger than $+$. For example, $a{\cdot}b + a{\cdot}c$ represents $(a{\cdot}b) + (a{\cdot}c)$. Occurrences of $\cdot$ are often omitted from process terms; that is, st denotes $s{\cdot}t$.

3.3 Bisimulation

In the previous section, each basic process term has been provided with a process graph using structural operational semantics. Processes have been studied since the early 60's, first to settle questions in natural languages, later on to study the semantics of programming languages. These studies originally focused on so-called trace equivalence, in which two processes are said to be equivalent if they can execute exactly the same strings of actions. However, for system behaviour this equivalence is not always satisfactory, which is shown by the following example.

Example 3. Consider the two processes below:

$$\begin{tikzcd}
& {} \arrow[d, "read(d)"'] & & & & {} \arrow[dl, "read(d)"'] \arrow[dr, "read(d)"] & \\
& {} \arrow[dl, "write_1(d)"'] \arrow[dr, "write_2(d)"] & & & {} \arrow[d, "write_1(d)"'] & & {} \arrow[d, "write_2(d)"] \\
\surd & & \surd & & \surd & & \surd
\end{tikzcd}$$

The first process reads datum d, and then decides whether it writes d on disc 1 or on disc 2. The second process makes a choice for disc 1 or disc 2 before it reads datum d. Both processes display the same strings of actions, $read(d)write_1(d)$ and $read(d)write_2(d)$, so they are trace equivalent. Still, there is a crucial distinction between the two processes, which becomes apparent if for instance disc 1 crashes. In this case the first process always saves datum d on disc 2, while the second process may get into a deadlock (i.e., may get stuck).

Bisimilarity, defined below, is more discriminative than trace equivalence. Namely, if two processes are bisimilar, then not only they can execute exactly the same strings of actions, but also they have the same branching structure. For example, the two processes in Example 3 are not bisimilar.

A bisimulation relation $\mathcal{B}$ is a binary relation on states in process graphs such that:

1. if $s\,\mathcal{B}\,t$ and $s \xrightarrow{a} s'$, then $t \xrightarrow{a} t'$ with $s'\,\mathcal{B}\,t'$;
2. if $s\,\mathcal{B}\,t$ and $t \xrightarrow{a} t'$, then $s \xrightarrow{a} s'$ with $s'\,\mathcal{B}\,t'$;
3. if $s\,\mathcal{B}\,t$ and $s \xrightarrow{a} \surd$, then $t \xrightarrow{a} \surd$;
4. if $s\,\mathcal{B}\,t$ and $t \xrightarrow{a} \surd$, then $s \xrightarrow{a} \surd$.

Two states s and t are bisimilar, denoted by $s \underline{\leftrightarrow}\, t$, if there is a bisimulation relation $\mathcal{B}$ such that $s\,\mathcal{B}\,t$.

Example 4. $(a+a)b \,\underline{\leftrightarrow}\, ab + a(b+b)$.
A bisimulation relation that relates these two basic process terms is defined by $(a+a)\,b\,\mathcal{B}\,ab + a(b+b)$, $b\,\mathcal{B}\,b$, and $b\,\mathcal{B}\,b+b$. This bisimulation relation can be depicted as follows:

$$\begin{tikzcd}
(a+a)b \arrow[d, bend right, "a"'] \arrow[d, bend left, "a"] \arrow[rr, dash, dashed] & & ab+a(b+b) \arrow[dl, "a"'] \arrow[d, "a"] \\
b \arrow[r, dash, dashed] \arrow[rr, dash, dashed, bend right] \arrow[d, "b"'] & b \arrow[dr, "b"'] & b+b \arrow[d, bend right, "b"'] \arrow[d, bend left, "b"] \\
\surd & & \surd
\end{tikzcd}$$

Bisimilarity is a congruence with respect to BPA. That is, if $s \,\underline{\leftrightarrow}\, s'$ and $t \,\underline{\leftrightarrow}\, t'$, then $s + t \,\underline{\leftrightarrow}\, s' + t'$ and $s{\cdot}t \,\underline{\leftrightarrow}\, s'{\cdot}t'$. This follows from the fact that the transition rules in Table 1 are in the so-called path format [21].

3.4 Axioms for BPA

Checking whether the process graphs of two basic process terms are bisimilar requires hard labour. First these process graphs have to be computed, and next a bisimulation relation has to be established between their root states. This section introduces an axiomatisation for BPA, to equate bisimilar basic process terms. This avoids the computation of process graphs and bisimulation relations altogether. The axioms have the additional advantage that they can be used in automated reasoning, so that they facilitate a mechanised derivation that two basic process terms are bisimilar.

We are after an axiomatisation such that the induced equality relation $=$ on basic process terms characterises bisimilarity over BPA in the following sense:

1. the equality relation is sound, meaning that if $s = t$ holds for basic process terms s and t, then $s \underline{\leftrightarrow} t$;
2. the equality relation is complete, meaning that if $s \underline{\leftrightarrow} t$ holds for basic process terms s and t, then $s = t$.

Soundness ensures that if terms can be equated, then they are in the same bisimilarity class, while completeness ensures that bisimilar terms can always be equated.

Table 2. Axioms for BPA

A1	$x + y = y + x$
A2	$(x + y) + z = x + (y + z)$
A3	$x + x = x$
A4	$(x + y){\cdot}z = x{\cdot}z + y{\cdot}z$
A5	$(x{\cdot}y){\cdot}z = x{\cdot}(y{\cdot}z)$

Table 2 presents an axiomatisation for BPA modulo bisimilarity. The equality relation on basic process terms induced by this axiomatisation is obtained by taking the set of substitution instances of A1-5, and closing it under equivalence and contexts.

The axiomatisation A1-5 is sound for BPA modulo bisimilarity. Since bisimilarity is both an equivalence and a congruence for BPA, it suffices to check the soundness of the individual axioms.

Moreover, the axiomatisation is complete for BPA modulo bisimilarity, meaning that $s \underline{\leftrightarrow} t$ implies $s = t$. This can be proved by directing axioms A3-5 from left to right, so that we obtain a term rewriting system (see, e.g., [20]). One can show that bisimilar basic process terms reduce to the same normal form, modulo the axioms A1,2.

From now on, process terms are considered modulo associativity of the $+$, and we often write $t_1 + t_2 + t_3$ instead of $(t_1 + t_2) + t_3$ or $t_1 + (t_2 + t_3)$.

4 Algebra of Communicating Processes

Atomic actions and the operators alternative and sequential composition from the previous section provide relatively primitive tools to construct a process graph. In general, the size of a basic process term is comparable to the size of the related process graph. This section introduces operators to express parallelism and concurrency, which enable us to capture a large process graph by means of a comparatively small process term.

4.1 Parallelism and Communication

In practice, process behaviour is often composed of several processors that are executed in parallel, where these separate entities may influence each other's execution. One could say that the processors are the building blocks that make up the complete system, cemented together by mutual communication actions. In order to model such concurrent systems, we introduce the merge, which is a binary operator that executes the two process terms in its arguments in parallel. That is, $s\|t$ can choose to execute an initial transition of s (i.e., a transition $s \xrightarrow{a} s'$ or $s \xrightarrow{a} \surd$) or an initial transition of t. This is formalised by four transition rules for the merge:

$$\frac{x \xrightarrow{v} \surd}{x\|y \xrightarrow{v} y} \qquad \frac{x \xrightarrow{v} x'}{x\|y \xrightarrow{v} x'\|y} \qquad \frac{y \xrightarrow{v} \surd}{x\|y \xrightarrow{v} x} \qquad \frac{y \xrightarrow{v} y'}{x\|y \xrightarrow{v} x\|y'}$$

Moreover, $s\|t$ can choose to execute a communication between initial transitions of s and t. For this purpose we assume a communication function $\gamma : A \times A \to A$, which produces for each pair of atomic actions a and b their communication $\gamma(a, b)$. This communication function is required to be commutative and associative; that is, for $a, b, c \in A$,

$$\gamma(a,b) \equiv \gamma(b,a)$$
$$\gamma(\gamma(a,b),c) \equiv \gamma(a,\gamma(b,c)).$$

The next four transition rules for the merge express that $s\|t$ can choose to execute a communication of initial transitions of s and t:

$$\frac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} \surd}{x\|y \xrightarrow{\gamma(v,w)} \surd} \qquad \frac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} y'}{x\|y \xrightarrow{\gamma(v,w)} y'} \qquad \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} \surd}{x\|y \xrightarrow{\gamma(v,w)} x'} \qquad \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} y'}{x\|y \xrightarrow{\gamma(v,w)} x'\|y'}$$

Example 5. Let the communication of two atomic actions from $\{a, b, c\}$ always result in c. The process graph of the process term $(ab)\|(ba)$ is depicted below.

This example shows that the merge of two simple process terms produces a relatively large process graph. This partly explains the strength of a theory of communicating processes, as this theory makes it possible to draw conclusions about the full system by studying its separate concurrent components.

$$\begin{tikzcd}
& & & & & & (ab)\|(ba) \arrow[dlll, "a"'] \arrow[d, "c"] \arrow[drrr, "b"] & & & & & & \\
& & & b\|(ba) \arrow[dl, "b"'] \arrow[d, "c"] \arrow[dr, "b"] & & & b\|a \arrow[dl, "b"'] \arrow[d, "c"] \arrow[dr, "a"] & & & (ab)\|a \arrow[dl, "a"'] \arrow[d, "c"] \arrow[dr, "a"] & & & \\
& & b\|a \arrow[dll, "b"'] \arrow[dl, "c"] \arrow[d, "a"] & a \arrow[d, "a"] & ba \arrow[d, "b"] & a \arrow[d, "a"] & \surd & b \arrow[d, "b"] & ab \arrow[d, "a"] & b \arrow[d, "b"] & b\|a \arrow[d, "b"] \arrow[dr, "c"] \arrow[drr, "a"] & & \\
a \arrow[d, "a"] & \surd & b \arrow[d, "b"] & \surd & a \arrow[d, "a"] & \surd & & \surd & b \arrow[d, "b"] & \surd & a \arrow[d, "a"] & \surd & b \arrow[d, "b"] \\
\surd & & \surd & & \surd & & & & \surd & & \surd & & \surd
\end{tikzcd}$$

4.2 Left Merge and Communication Merge

Moller [16] proved that there does not exist a sound and complete finite axiomatisation for BPA extended with the merge, modulo bisimilarity. This problem is overcome by defining two extra operators, called left merge and communication merge, which both capture part of the behaviour of the merge.

The left merge $s \mathbin{\lfloor\!\lfloor} t$ takes its initial transition from the process term s, and then behaves as the merge $\|$. This is expressed by two transition rules for the left merge, which correspond with the first two transition rules for the merge:

$$\frac{x \xrightarrow{v} \surd}{x \mathbin{\lfloor\!\lfloor} y \xrightarrow{v} y} \qquad \frac{x \xrightarrow{v} x'}{x \mathbin{\lfloor\!\lfloor} y \xrightarrow{v} x' \| y}$$

The communication merge $s|t$ executes as initial transition a communication between initial transitions of the process terms s and t, and then behaves as the merge $\|$. This is expressed by four transition rules for the communication merge, which correspond with the last four transition rules for the merge:

$$\frac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} \surd}{x|y \xrightarrow{\gamma(v,w)} \surd} \qquad \frac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} y'}{x|y \xrightarrow{\gamma(v,w)} y'} \qquad \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} \surd}{x|y \xrightarrow{\gamma(v,w)} x'} \qquad \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} y'}{x|y \xrightarrow{\gamma(v,w)} x' \| y'}$$

As binding convention we assume that $\|$, $\mathbin{\lfloor\!\lfloor}$, and $|$ bind stronger than $+$. For example, $a \mathbin{\lfloor\!\lfloor} b + a\|c$ represents $(a \mathbin{\lfloor\!\lfloor} b) + (a\|c)$. We refer to BPA extended with the three parallel operators $\|$, $\mathbin{\lfloor\!\lfloor}$, and $|$ as PAP (for process algebra with parallelism).

The left and communication merge together cover the behaviour of the merge, in the sense that $s\|t \;\underline{\leftrightarrow}\; (s \mathbin{\lfloor\!\lfloor} t + t \mathbin{\lfloor\!\lfloor} s) + s|t$. Namely, $s\|t$ can execute either an initial transition of s or t, which is covered by $s \mathbin{\lfloor\!\lfloor} t$ or $t \mathbin{\lfloor\!\lfloor} s$, respectively, or a communication of initial transitions of s and t, which is covered by $s|t$.

The transition rules of PAP constitute a conservative extension of the ones of BPA, meaning that they do not influence the process graphs of basic process terms. That is, an initial transition of a basic process term is derivable from the transition rules of PAP if and only if this transition can be derived from the transition rules of BPA. This follows from the fact that this extension adheres to the syntactic restrictions of the conservative extension format from [11].

Bisimilarity is a congruence with respect to PAP. Again this follows from the fact that the transition rules of PAP are in the path format.

4.3 Axioms for PAP

Table 3 presents the axioms for the three parallel operators modulo bisimilarity. We already noted that the merge can be split into the left merge and the communication merge; this is exploited in axiom M1. Axioms LM2-4 and CM5-10 enable us to eliminate occurrences of the left merge and the communication merge from process terms. The axioms for PAP are added to the ones for BPA.

Table 3. Axioms for merge, left merge, and communication merge

M1	$x \| y = (x ⫦ y + y ⫦ x) + x|y$
LM2	$v ⫦ y = v{\cdot}y$
LM3	$(v{\cdot}x) ⫦ y = v{\cdot}(x \| y)$
LM4	$(x+y) ⫦ z = x ⫦ z + y ⫦ z$
CM5	$v|w = \gamma(v,w)$
CM6	$v|(w{\cdot}y) = \gamma(v,w){\cdot}y$
CM7	$(v{\cdot}x)|w = \gamma(v,w){\cdot}x$
CM8	$(v{\cdot}x)|(w{\cdot}y) = \gamma(v,w){\cdot}(x \| y)$
CM9	$(x+y)|z = x|z + y|z$
CM10	$x|(y+z) = x|y + x|z$

It can be proved that the resulting axiomatisation is sound and complete for PAP modulo bisimilarity. Again, the completeness proof is based on a term rewriting analysis, in which the axioms are directed from left to right.

4.4 Deadlock and Encapsulation

If two atomic actions are able to communicate, then often we only want these actions to occur in communication with each other, and not on their own. For example, let the action $send(d)$ represent sending a datum d into one end of a channel, while $read(d)$ represents receiving this datum at the other end of the channel. Furthermore, let the communication of these two actions result in transferring the datum d through the channel by the action $comm(d)$. For the outside world, the actions $send(d)$ and $read(d)$ never appear on their own, but only in communication in the form $comm(d)$.

In order to enforce communication in such cases, we introduce a special constant δ called deadlock, which does not display any behaviour. The communication function γ

is extended by allowing that the communication of two atomic actions results in δ, i.e., $\gamma : A \times A \rightarrow A \cup \{\delta\}$. This extension of γ enables us to express that two actions a and b do not communicate, by defining $\gamma(a, b) \equiv \delta$. Furthermore, we introduce unary encapsulation operators ∂_H for sets H of atomic actions, which rename all actions in H into δ. PAP extended with deadlock and encapsulation operators is called the algebra of communicating processes (ACP).

Since the deadlock does not display any behaviour, there is no transition rule for this constant. Furthermore, since the communication of actions can result in δ, the last four transition rules for the merge and the four transition rules for the communication merge need to be supplied with the requirement $\gamma(v, w) \not\equiv \delta$. Finally, the behaviour of the encapsulation operators is captured by the following transition rules, which express that $\partial_H(t)$ can execute those transitions of t that have a label outside H:

$$\frac{x \xrightarrow{v} \surd}{\partial_H(x) \xrightarrow{v} \surd}\; v \notin H \qquad\qquad \frac{x \xrightarrow{v} x'}{\partial_H(x) \xrightarrow{v} \partial_H(x')}\; v \notin H$$

We give an example of the use of encapsulation operators.

Example 6. Suppose a datum 0 or 1 is sent into a channel, which is expressed by the process term $\mathit{send}(0) + \mathit{send}(1)$. Let this datum be received at the other side of the channel, which is expressed by the process term $\mathit{read}(0) + \mathit{read}(1)$. The communication of $\mathit{send}(d)$ and $\mathit{read}(d)$ results in $\mathit{comm}(d)$ for $d \in \{0, 1\}$, while all other communications between actions result in δ. The behaviour of the channel is described by the process term

$$\partial_{\{\mathit{send}(0),\, \mathit{send}(1),\, \mathit{read}(0),\, \mathit{read}(1)\}}((\mathit{send}(0) + \mathit{send}(1)) \| (\mathit{read}(0) + \mathit{read}(1)))$$

The encapsulation operator enforces that the action $\mathit{send}(d)$ can only occur in communication with the action $\mathit{read}(d)$, for $d \in \{0, 1\}$.

Beware not to confuse a transition of the form $t \xrightarrow{a} \delta$ with a transition of the form $t \xrightarrow{a} \surd$; intuitively, the first transition expresses that t gets stuck after the execution of a, while the second transition expresses that t terminates successfully after the execution of a. A process term t is said to contain a deadlock if there are transitions $t \xrightarrow{a_1} t_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} t_n$ such that the process term t_n does not have any initial transitions (i.e., $t_n \underline{\leftrightarrow} \delta$). In general it is undesirable that a process contains a deadlock, because it represents that the process gets stuck without producing any output. Experience learns that non-trivial specifications of system behaviour often contain a deadlock. For example, the third sliding window protocol in [19] contained a deadlock; see [14, *Stelling 7*]. It can, however, be very difficult to detect such a deadlock, even if one has a good insight into such a protocol. Automated tools have been developed to help with the detection of deadlocks in a process algebraic framework.

ACP is a conservative extension of PAP, meaning that the transition rules for the encapsulation operators do not influence the process graphs belonging to process terms in PAP. Again this follows from the fact that this extension adheres to the syntactic restrictions of the conservative extension format. Moreover, bisimilarity is a congruence with respect to ACP, because the transition rules of ACP are in the path format.

Table 4. Axioms for deadlock and encapsulation

A6		$x + \delta = x$
A7		$\delta \cdot x = \delta$
D1	$v \notin H$	$\partial_H(v) = v$
D2	$v \in H$	$\partial_H(v) = \delta$
D3		$\partial_H(\delta) = \delta$
D4		$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$
D5		$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$
LM11		$\delta \llfloor x = \delta$
CM12		$\delta \mid x = \delta$
CM13		$x \mid \delta = \delta$

Table 4 presents axioms A6,7 for the deadlock, axioms D1-5 for encapsulation, and axioms LM11 and CM12,13 to deal with the interplay of the deadlock with left and communication merge.

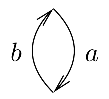
It can be proved that the resulting axiomatisation is sound and complete for ACP modulo bisimilarity. Again, the completeness proof is based on a term rewriting analysis, in which the axioms are directed from left to right.

5 Recursion

Up to now we have focused on finite processes. However, systems can often exhibit infinite traces. In this section it is shown how such infinite behaviour can be specified using recursive equations.

5.1 Guarded Recursive Specifications

Consider the process that alternately executes actions a and b until infinity, with the root node presented at the top:



Since ACP can only specify finite behaviour, there does not exist a process term in ACP with this (or a bisimilar) process graph. The process above can be captured by means of two recursive equations:

$$\begin{aligned} X &= aY \\ Y &= bX. \end{aligned}$$

Here, X and Y are recursion variables, which intuitively represent the two states of the process in which it is going to execute a or b, respectively.

In general, a recursive specification consists of a finite set of recursive equations

$$\begin{aligned} X_1 &= t_1(X_1,\ldots,X_n)\\ &\vdots\\ X_n &= t_n(X_1,\ldots,X_n) \end{aligned}$$

where the left-hand sides X_i are recursion variables, and the $t_i(X_1,\ldots,X_n)$ at the right-hand sides are process terms in ACP with possible occurrences of the recursion variables $X_1,\ldots,X_n$.

Process terms $s_1,\ldots,s_n$ are said to be a solution for a recursive specification $\{X_i = t_i(X_1,\ldots,X_n) \mid i \in \{1,\ldots,n\}\}$ (with respect to bisimilarity) if $s_i \mathrel{\underline{\leftrightarrow}} t_i(s_1,\ldots,s_n)$ for all $i \in \{1,\ldots,n\}$.

A recursive specification should represent a unique process graph, so we want its solution to be unique, modulo bisimilarity. That is, if $s_1,\ldots,s_n$ and $s_1',\ldots,s_n'$ are two solutions for the same recursive specification, then $s_i \mathrel{\underline{\leftrightarrow}} s_i'$ for $i \in \{1,\ldots,n\}$. However, there exist recursive specifications that allow more than one solution modulo bisimilarity. We give some examples.

Example 7. Let $a \in A$.

1. All process terms are a solution for the recursive specification $\{X{=}X\}$.
2. All process terms s that can execute an initial transition $s \xrightarrow{a} \surd$ are a solution for the recursive specification $\{X{=}a{+}X\}$.
3. All process terms that cannot terminate successfully are a solution for the recursive specification $\{X{=}Xa\}$.

The following example features recursive specifications that do have a unique solution modulo bisimilarity.

Example 8. Let $a, b \in A$.

1. The only solution for $\{X{=}aY, Y{=}bX\}$, modulo bisimilarity, is $X \mathrel{\underline{\leftrightarrow}} abab\cdots$ and $Y \mathrel{\underline{\leftrightarrow}} baba\cdots$.
2. The only solution for $\{X{=}Y, Y{=}aX\}$, modulo bisimilarity, is $X \mathrel{\underline{\leftrightarrow}} aaa\cdots$ and $Y \mathrel{\underline{\leftrightarrow}} aaa\cdots$.
3. The only solution for $\{X{=}(a{+}b) \mathbin{\lfloor\!\lfloor} X\}$, modulo bisimilarity, is $X \mathrel{\underline{\leftrightarrow}} (a{+}b)(a{+}b)$ $(a+b)\cdots$.

A recursive specification allows a unique solution modulo bisimilarity if and only if it is guarded. A recursive specification

$$\begin{aligned} X_1 &= t_1(X_1,\ldots,X_n)\\ &\vdots\\ X_n &= t_n(X_1,\ldots,X_n) \end{aligned}$$

is guarded if the right-hand sides of its recursive equations can be adapted to the form

$$a_1{\cdot}s_1(X_1,\ldots,X_n) + \cdots + a_k{\cdot}s_k(X_1,\ldots,X_n) + b_1 + \cdots + b_\ell$$

with $a_1, \ldots, a_k, b_1, \ldots, b_\ell \in A$, by applications of the axioms of ACP and replacing recursion variables by the right-hand sides of their recursive equations. The process term above is allowed to have zero summands (i.e., k and ℓ can both be zero), in which case it represents the deadlock δ.

The recursive specifications in Example 7 are all unguarded; that is, their right-hand sides cannot be brought into the desired form presented above. The recursive specifications in Example 8 are all guarded.

5.2 Transition Rules for Guarded Recursion

If E is a guarded recursive specification, and X a recursion variable in E, then intuitively $\langle X|E\rangle$ denotes the process that has to be substituted for X in the solution for E. For instance, if E is $\{X{=}aY, Y{=}bX\}$, then $\langle X|E\rangle$ represents the process $abab\cdots$, while $\langle Y|E\rangle$ represents the process $baba\cdots$; see the first recursive specification in Example 8. We extend ACP with the constants $\langle X|E\rangle$, for guarded recursive specifications E and recursion variables X in E.

Assume that the guarded recursive specification E is of the form

$$\begin{aligned} X_1 &= t_1(X_1, \ldots, X_n) \\ &\vdots \\ X_n &= t_n(X_1, \ldots, X_n). \end{aligned}$$

Guarded recursion is captured by two transition rules which express that the behaviour of the solutions $\langle X_i|E\rangle$ for the recursion variables X_i in E, for each $i \in \{1, \ldots, n\}$, is exactly the behaviour of its right-hand side $t_i(X_1, \ldots, X_n)$:

$$\frac{t_i(\langle X_1|E\rangle, \ldots, \langle X_n|E\rangle) \xrightarrow{v} \surd}{\langle X_i|E\rangle \xrightarrow{v} \surd} \qquad \frac{t_i(\langle X_1|E\rangle, \ldots, \langle X_n|E\rangle) \xrightarrow{v} y}{\langle X_i|E\rangle \xrightarrow{v} y}$$

Example 9. Let E denote $\{X{=}aY, Y{=}bX\}$. The process graph of $\langle X|E\rangle$ is

$$\begin{tikzcd} \langle X|E\rangle \arrow[d, bend left, "a"] \\ \langle Y|E\rangle \arrow[u, bend left, "b"] \end{tikzcd}$$

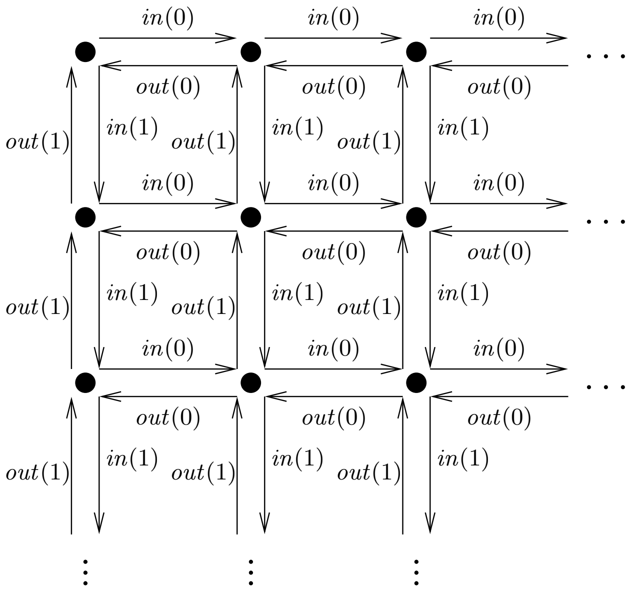
The transition $\langle X|E\rangle \xrightarrow{a} \langle Y|E\rangle$ can be derived from the transition rules as follows:

$$\begin{array}{cll}
a \xrightarrow{a} \surd & \left(\dfrac{}{v \xrightarrow{v} \surd},\right. & \left. v := a\right) \\
\cline{1-1}
a\langle Y|E\rangle \xrightarrow{a} \langle Y|E\rangle & \left(\dfrac{x \xrightarrow{v} \surd}{xy \xrightarrow{v} y},\right. & \left. v := a,\ x := a,\ y := \langle Y|E\rangle\right) \\
\cline{1-1}
\langle X|E\rangle \xrightarrow{a} \langle Y|E\rangle & \left(\dfrac{a\langle Y|E\rangle \xrightarrow{v} y}{\langle X|E\rangle \xrightarrow{v} y},\right. & \left. v := a,\ y := \langle Y|E\rangle\right)
\end{array}$$

ACP with guarded recursion is a conservative extension of ACP, because this extension adheres to the syntactic restrictions of the conservative extension format. Moreover, bisimilarity is a congruence with respect to ACP with guarded recursion, because the transition rules of guarded recursion are in the path format.

As an example of the use of guarded recursion we consider the bag process over the set $\{0,1\}$.

Example 10. We specify a process that can put elements 0 and 1 into a bag, and subsequently collect these elements from the bag in arbitrary order. The actions $in(0)$ and $in(1)$ represent putting a 0 or 1 into the bag, respectively. Similarly, the actions $out(0)$ and $out(1)$ represent collecting a 0 or 1 from the bag, respectively. All communications between actions result in δ. Initially the bag is empty, so that one can only put an element into the bag. The process graph below depicts the behaviour of the bag over $\{0,1\}$, with the root state placed in the leftmost uppermost corner. Note that this bag process consists of infinitely many non-bisimilar states.



The bag over $\{0,1\}$ can be specified by a single recursive equation, using the merge $\|$. Let E denote the guarded recursive specification

$$X \;=\; in(0)\cdot(X\|out(0)) + in(1)\cdot(X\|out(1)).$$

The process graph of $\langle X|E\rangle$ is bisimilar with the behaviour of the bag over $\{0,1\}$ as depicted above. Namely, initially $\langle X|E\rangle$ can only execute an action $in(d)$ for $d \in \{0,1\}$. The subsequent process term $\langle X|E\rangle\|out(d)$ can put elements 0 and 1 in the bag and take them out again (by means of the parallel component $\langle X|E\rangle$), or it can at any time take the initial element d out of the bag (by means of the parallel component $out(d)$).

5.3 Recursive Definition and Specification Principles

As before, we want to fit guarded recursion into an axiomatic framework. Table 5 contains two axioms for guarded recursion, the recursive definition principle (RDP) and the recursive specification principle (RSP). The guarded recursive specification E in the axioms is assumed to be of the form

$$\begin{aligned} X_1 &= t_1(X_1, \ldots, X_n) \\ &\vdots \\ X_n &= t_n(X_1, \ldots, X_n). \end{aligned}$$

Intuitively, RDP expresses that $\langle X_1|E\rangle, \ldots, \langle X_n|E\rangle$ is a solution for E, while RSP expresses that this is the only solution for E modulo bisimilarity.

Table 5. Recursive definition and specification principles

RDP	$\langle X_i\|E\rangle = t_i(\langle X_1\|E\rangle, \ldots, \langle X_n\|E\rangle)$	$(i \in \{1, \ldots, n\})$
RSP	If $y_i = t_i(y_1, \ldots, y_n)$ for all $i \in \{1, \ldots, n\}$, then	
	$y_i = \langle X_i\|E\rangle$	$(i \in \{1, \ldots, n\})$

The resulting axomatisation is sound for ACP with guarded recursion modulo bisimilarity. However, it is not complete. For instance, the following two symmetric guarded recursive specifications of the bag over $\{0, 1\}$ (see Example 10) are bisimilar, but cannot be proved equal by means of the axioms:

$$\begin{aligned} X &= in(0)\cdot(X \| out(0)) + in(1)\cdot(X \| out(1)) \\ Y &= in(0)\cdot(out(0) \| Y) + in(1)\cdot(out(1) \| Y). \end{aligned}$$

(In this particular case, this could be remedied by adding a commutativity axiom for the merge.)

One can prove that the axiomatisation is complete for the subclass of linear recursive specifications. A recursive specification is linear if its recursive equations are of the form

$$X = a_1 X_1 + \cdots + a_k X_k + b_1 + \cdots + b_\ell$$

with $a_1, \ldots, a_k, b_1, \ldots, b_\ell \in A$. (The empty sum represents δ.) Note that a linear recursive specification is by default guarded.

A regular process, which by definition consists of finitely many states and transitions, can always be described by a linear recursive specification. Namely, each state s in the regular process can be represented by a recursion variable X_s. If state s can evolve into state s' by the execution of an action a, then this is expressed by a summand $aX_{s'}$ at the right-hand side of the recursive equation for X_s. Moreover, if state s can terminate successfully by the execution of an action a, then this is expressed by a summand a at the right-hand side of the recursive equation for X_s. The result is a linear recursive specification E, and $\langle X_s|E\rangle \underline{\leftrightarrow}\ s$ for all states s in the regular process. Vice versa, a linear recursive specification always gives rise to a regular process.

6 Abstraction

If a customer asks a programmer to implement a product, ideally this customer is able to provide the external behaviour of the desired program. That is, he or she is able to tell what should be the output of the program for each possible input. The programmer then comes up with an implementation. The question is, does this implementation really display the desired external behaviour? To answer this question, we need to abstract away from the internal computation steps of the program.

6.1 Rooted Branching Bisimulation

In order to abstract away from internal actions, we introduce a special constant τ, called the silent step. Intuitively, a τ-transition represents a sequence of internal actions that can be eliminated from a process graph. As any atomic action, the constant τ can execute itself, after which it terminates successfully. This is expressed by the transition rule

$$\frac{}{\tau \xrightarrow{\tau} \surd}$$

From now on, v and w in the transition rules and the axioms of ACP with guarded recursion range over $A \cup \{\tau\}$. (So the transition rule for atomic actions in Table 1 yields the transition rule for the silent step τ presented above.) The domain of the communication function γ is extended with the silent step, $\gamma : A \cup \{\tau\} \times A \cup \{\tau\} \to A \cup \{\delta\}$, by defining that each communication involving τ results in δ.

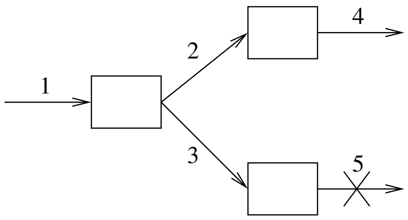
In the presence of the silent step τ, bisimilarity is no longer a satisfactory process equivalence. Namely, if process terms s and t are equivalent, and s can execute an action τ, then it need not be the case that t can simulate this τ-transition of s by the execution of an action τ. The intuition for the silent step, that it represents an internal computation in which we are not really interested, asks for a new process equivalence. The question that we must pose ourselves is:

which τ-transitions are truly silent?

The obvious answer to this question, "all τ-transitions are truly silent", turns out to be incorrect. Namely, this answer would produce an equivalence relation that does not preserve deadlock behaviour.

As an example of an action τ that is not truly silent, consider the process terms $a + \tau\delta$ and a. If the τ in the first term were truly silent, then these two terms would be equivalent. However, the process graph of the first term contains a deadlock, $a+\tau\delta \xrightarrow{\tau} \delta$, while the process graph of the second term does not. Hence, the τ in the first term is not truly silent. In order to describe this case more vividly, we give an example.

Example 11. Consider a protocol that first receives a datum d via channel 1, and then communicates this datum via channel 2 or via channel 3. If the datum is communicated through channel 2, then it is sent into channel 4. If the datum is communicated through channel 3, then it gets stuck, as the subsequent channel 5 is broken. So the system gets into a deadlock if the datum d is transferred via channel 3. This deadlock should not disappear if we abstract away from the internal communication actions via channels 2 and 3, because this would cover up an important problem of the protocol.



The system, which is depicted above, is described by the process term

$$\partial_{\{s_5(d)\}}(r_1(d)\cdot(c_2(d)\cdot s_4(d) + c_3(d)\cdot s_5(d))) \overset{\mathrm{D1,2,4,5}}{=} r_1(d)\cdot(c_2(d)\cdot s_4(d) + c_3(d)\cdot\delta)$$

where $s_i(d)$, $r_i(d)$, and $c_i(d)$ represent a send, read, and communication action of the datum d via channel i, respectively. Abstracting away from the internal actions $c_2(d)$ and $c_3(d)$ in this process term yields $r_1(d)\cdot(\tau\cdot s_4(d)+\tau\cdot\delta)$. The second τ in this process term cannot be deleted, because then the process would no longer be able to get into a deadlock. Hence, this τ is not truly silent.

As a further example of a τ-transition that is not truly silent, consider the process terms $a + \tau b$ and $a + b$. We argued previously that the process terms $\partial_{\{b\}}(a + \tau b) = a + \tau\delta$ and $\partial_{\{b\}}(a+b) = a$ are not equivalent, because the first term contains a deadlock while the second term does not. Hence, $a + \tau b$ and $a + b$ cannot be equivalent, for else the envisioned equivalence relation would not be a congruence.

Problems with deadlock preservation and congruence can be avoided by taking a more restrictive view on abstracting away from silent steps. A correct answer to the question

which τ-transitions are truly silent?

turns out to be

those τ-transitions that do not lose possible behaviours!

For example, the process terms $a + \tau(a + b)$ and $a + b$ are equivalent, because the τ in the first process term is truly silent: after execution of this τ it is still possible to execute a. In general, process terms $s + \tau(s + t)$ and $s + t$ are equivalent for all process terms s and t. By contrast, in a process term such as $a + \tau b$ the τ is not truly silent, since execution of this τ means losing the option to execute a.

The intuition above is formalised in the notion of branching bisimilarity. Let the process terms s and t be branching bisimilar. If $s \xrightarrow{\tau} s'$, then t does not have to simulate this τ-transition if it is truly silent, meaning that s' and t are branching bisimilar. Moreover, a non-silent transition $s \xrightarrow{a} s'$ need not be simulated by t immediately, but only after a number of truly silent τ-transitions: $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_0 \xrightarrow{a} t'$, where s and t_0 are branching bisimilar (to ensure that the τ-transitions are truly silent) and s' and t' are branching bisimilar (so that $s \xrightarrow{a} s'$ is simulated by $t_0 \xrightarrow{a} t'$). A special termination predicate $\downarrow$ is needed in order to relate branching bisimilar process terms such as $a\tau$ and a.

Assume a special termination predicate $\downarrow$, and let $\surd$ represent a state with $\surd \downarrow$. A branching bisimulation relation $\mathcal{B}$ is a binary relation on states in process graphs such that:

1. if $s\,\mathcal{B}\,t$ and $s \xrightarrow{a} s'$, then
   - either $a \equiv \tau$ and $s'\,\mathcal{B}\,t$;
   - or there is a sequence of (zero or more) τ-transitions $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_0$ such that $s\,\mathcal{B}\,t_0$ and $t_0 \xrightarrow{a} t'$ with $s'\,\mathcal{B}\,t'$;
2. if $s\,\mathcal{B}\,t$ and $t \xrightarrow{a} t'$, then
   - either $a \equiv \tau$ and $s\,\mathcal{B}\,t'$;
   - or there is a sequence of (zero or more) τ-transitions $s \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_0$ such that $s_0\,\mathcal{B}\,t$ and $s_0 \xrightarrow{a} s'$ with $s'\,\mathcal{B}\,t'$.
3. if $s\,\mathcal{B}\,t$ and $s \downarrow$, then there is a sequence of (zero or more) τ-transitions $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_0$ such that $s\,\mathcal{B}\,t_0$ and $t_0 \downarrow$;
4. if $s\,\mathcal{B}\,t$ and $t \downarrow$, then there is a sequence of (zero or more) τ-transitions $s \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_0$ such that $s_0\,\mathcal{B}\,t$ and $s_0 \downarrow$.

Two states s and t are branching bisimilar, denoted by $s \underline{\leftrightarrow}_b\, t$, if there is a branching bisimulation relation $\mathcal{B}$ such that $s\,\mathcal{B}\,t$.

Example 12. $a + \tau(a+b) \underline{\leftrightarrow}_b\, \tau(a+b) + b$.
A branching bisimulation relation that relates these two process terms is defined by $a+\tau(a+b)\,\mathcal{B}\,\tau(a+b)+b$, $a+b\,\mathcal{B}\,\tau(a+b)+b$, $a+\tau(a+b)\,\mathcal{B}\,a+b$, $a+b\,\mathcal{B}\,a+b$, and $\surd\,\mathcal{B}\,\surd$. This relation can be depicted as follows:

$$\begin{tikzcd}
 & a+\tau(a+b) \arrow[dl, "a"'] \arrow[d, "\tau"] \arrow[r, dash, dashed] \arrow[dr, dash, dashed] & \tau(a+b)+b \arrow[d, "\tau"'] \arrow[dr, "b"] \arrow[dl, dash, dashed] & \\
\surd & a+b \arrow[l, shift right, "a"'] \arrow[l, shift left, "b"] \arrow[r, dash, dashed] & a+b \arrow[r, shift left, "a"] \arrow[r, shift right, "b"'] & \surd \arrow[lll, dash, dashed, bend left]
\end{tikzcd}$$

It is left to the reader to verify that this relation satisfies the requirements of a branching bisimulation relation.

Branching bisimilarity satisfies a notion of *fairness*. That is, if an exit from a τ-loop exists, then no infinite execution sequence will remain in this τ-loop forever. The intuition is that there is zero chance that no exit from the τ-loop will ever be chosen. For example, it is not hard to see that $\langle X \mid X = \tau X + a\rangle$ and a are branching bisimilar.

Branching bisimilarity preserves a large class of interesting properties (including deadlock behaviour) [7]. See [13] for an exposition on why branching bisimilarity constitutes a sensible equivalence relation to abstract away from internal computations.

Branching bisimilarity is an equivalence relation; see [4]. However, it is still not a congruence with respect to BPA. For example, b and τb are branching bisimilar, but we already argued that $a + b$ and $a + \tau b$ are not branching bisimilar. This problem can be overcome by adding a rootedness condition: initial τ-transitions are never truly silent. In other words, two states are considered equivalent if they can simulate each other's initial transitions, such that the resulting states are branching bisimilar. This leads to the notion of rooted branching bisimilarity.

A rooted branching bisimulation relation $\mathcal{B}$ is a binary relation on states in process graphs such that:

1. if $s\,\mathcal{B}\,t$ and $s \xrightarrow{a} s'$, then $t \xrightarrow{a} t'$ with $s' \underline{\leftrightarrow}_b t'$;
2. if $s\,\mathcal{B}\,t$ and $t \xrightarrow{a} t'$, then $s \xrightarrow{a} s'$ with $s' \underline{\leftrightarrow}_b t'$;
3. if $s\,\mathcal{B}\,t$ and $s \downarrow$, then $t \downarrow$;
4. if $s\,\mathcal{B}\,t$ and $t \downarrow$, then $s \downarrow$.

Two states s and t are rooted branching bisimilar, denoted by $s \underline{\leftrightarrow}_{rb} t$, if there is a rooted branching bisimulation relation $\mathcal{B}$ such that $s\,\mathcal{B}\,t$.

Since branching bisimilarity is an equivalence relation, it is not hard to see that rooted branching bisimilarity is also an equivalence relation. Branching bisimilarity includes rooted branching bisimilarity, which in turn includes bisimilarity:

$$\underline{\leftrightarrow} \subset \underline{\leftrightarrow}_{rb} \subset \underline{\leftrightarrow}_b \,.$$

In the absence of τ (for example, in ACP), bisimilarity and branching bisimilarity induce exactly the same equivalence classes. In other words, two process terms in ACP are bisimilar if and only if they are branching bisimilar.

6.2 Guarded Linear Recursion Revisited

Assume a recursive specification E that consists of linear recursive equations $X_i = t_i(X_1, \ldots, X_n)$ for $i \in \{1, \ldots, n\}$. Since from now on we consider process terms in the setting of rooted branching bisimilarity, process terms $s_1, \ldots, s_n$ are said to be a solution for E (with respect to rooted branching bisimilarity) if $s_i \underline{\leftrightarrow}_{rb} t_i(s_1, \ldots, s_n)$ for $i \in \{1, \ldots, n\}$.

In the setting with the silent step, the notion of guardedness, which aims to classify those recursive specifications that have a unique solution modulo the process equivalence under consideration, needs to be adapted. For example, all process terms τs are solutions for the recursive specification $X = \tau X$, because $\tau s \underline{\leftrightarrow}_{rb} \tau\tau s$ holds for all process terms s. Hence, we consider such a recursive specification to be unguarded. The notion of guardedness is extended to linear recursive specifications that involve silent steps by requiring the absence of τ-loops.

A recursive specification is linear if its recursive equations are of the form

$$X \;=\; a_1X_1 + \cdots + a_kX_k + b_1 + \cdots + b_\ell$$

with $a_1, \ldots, a_k, b_1, \ldots, b_\ell \in A \cup \{\tau\}$. A linear recursive specification E is guarded if there does not exist an infinite sequence of τ-transitions $\langle X|E\rangle \xrightarrow{\tau} \langle X'|E\rangle \xrightarrow{\tau}$

$\langle X''|E\rangle \xrightarrow{\tau} \cdots$. The guarded linear recursive specifications are exactly the linear recursive specifications that have a unique solution, modulo rooted branching bisimilarity.

ACP with silent step guarded linear recursion constitutes a conservative extension of ACP with linear recursion, because this extension adheres to the syntactic restrictions of the conservative extension format. Moreover, rooted branching bisimilarity is a congruence with respect to ACP silent step and guarded linear recursion. This follows from the fact that the transition rules are in the RBB cool format from [9].

Table 6 presents the axioms B1,2 for the silent step, modulo rooted branching bisimilarity.

Table 6. Axioms for the silent step

B1	$v{\cdot}\tau = v$
B2	$v{\cdot}(\tau{\cdot}(x+y)+x) = v{\cdot}(x+y)$

The resulting axiomatisation is sound for ACP with silent step and guarded linear recursion, modulo rooted branching bisimilarity. Moreover, it can be shown that the axiomatisation is complete, see [12].

6.3 Abstraction Operators

We introduce unary abstraction operators τ_I, for subsets I of A, which rename all atomic actions in I into τ. The abstraction operators enable us to abstract away from the internal computation steps of an implementation. The behaviour of the abstraction operators is captured by the following transition rules, which express that in $\tau_I(t)$ all labels of transitions of t that are in I are renamed into τ:

$$\frac{x \xrightarrow{v} \surd}{\tau_I(x) \xrightarrow{v} \surd}\; v \notin I \qquad \frac{x \xrightarrow{v} x'}{\tau_I(x) \xrightarrow{v} \tau_I(x')}\; v \notin I$$

$$\frac{x \xrightarrow{v} \surd}{\tau_I(x) \xrightarrow{\tau} \surd}\; v \in I \qquad \frac{x \xrightarrow{v} x'}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')}\; v \in I$$

ACP extended with silent step and abstraction operators is denoted by ACP$_\tau$.

ACP$_\tau$ once again constitutes a conservative extension of ACP, because this extension adheres to the syntactic restrictions of the conservative extension format. Moreover, rooted branching bisimilarity is a congruence with respect to ACP$_\tau$ with guarded linear recursion, because the transition rules are in the RBB cool format.

Table 7 presents axioms for the abstraction operators, modulo rooted branching bisimilarity.

The resulting axiomatisation is sound for ACP$_\tau$ with guarded linear recursion modulo rooted branching bisimilarity. However, to obtain a complete axiomatisation, we need one more proof principle.

Table 7. Axioms for abstraction operators

TI1	$v \notin I$	$\tau_I(v) = v$
TI2	$v \in I$	$\tau_I(v) = \tau$
TI3		$\tau_I(\delta) = \delta$
TI4		$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$
TI5		$\tau_I(x{\cdot}y) = \tau_I(x){\cdot}\tau_I(y)$

6.4 Cluster Fair Abstraction Rule

Although τ-loops are prohibited in guarded linear recursive specifications, they can be constructed using an abstraction operator. For example, $\tau_{\{a\}}(\langle X \mid X{=}aX\rangle)$ can only execute τ's until infinity. This observation motivates the following distinction between specifiable and constructible regular processes:

– specifiable regular processes are the process graphs belonging to process terms in ACP with silent step and guarded linear recursion;
– constructible regular processes are the process graphs belonging to process terms in ACP$_\tau$ with guarded linear recursion.

$\tau\tau\tau\cdots$ is the simplest example of a regular process that is constructible, being the process graph of $\tau_{\{a\}}(\langle X \mid X{=}aX\rangle)$, but not specifiable. In general, a constructible regular process is specifiable if and only if it is free of τ-loops. One extra axiom is needed to equate process terms of which the regular process graphs are constructible but not specifiable. For example,

$$\tau_{\{a\}}(\langle X \mid X{=}aX\rangle) \underline{\leftrightarrow}_{rb}\ \tau_{\{a,b\}}(\langle Y \mid Y{=}aZ, Z{=}bY\rangle)$$

because both process terms execute τ's until infinity. However, these process terms cannot be equated by means of the axioms, due to the guardedness restriction on RSP, which is essential for the soundness of this axiom. In order to get rid of τ-loops, we introduce the notion of fair abstraction. For example, let E denote the following guarded linear recursive specification:

$$\begin{aligned} X_1 &= aX_2 + s_1 \\ &\ \vdots \\ X_{n-1} &= aX_n + s_{n-1} \\ X_n &= aX_1 + s_n \end{aligned}$$

for some $a \in A$. The process term $\tau_{\{a\}}(\langle X_1|E\rangle)$ executes τ-transitions that are the result of abstracting away from the occurrences of a in front of the recursion variables X_i, until it exits this τ-loop by executing one of the process terms $\tau_{\{a\}}(s_i)$ for $i \in \{1,\ldots,n\}$. Note that the transitions in the τ-loop are all truly silent, because they do not lose possible behaviours; after the execution of such a τ, it is still possible to execute any of the process terms $\tau_{\{a\}}(s_i)$ for $i \in \{1,\ldots,n\}$. Fair abstraction says that $\tau_{\{a\}}(\langle X_1|E\rangle)$ does not stay in the τ-loop forever, so that at some time it will start executing a $\tau_{\{a\}}(s_i)$. Hence,

$$\tau_{\{a\}}(\langle X_1|E\rangle) \underline{\leftrightarrow}_{rb}\ \tau_{\{a\}}(s_1 + \tau(s_1 + \cdots + s_n)).$$

Namely, initially $\tau_{\{a\}}(\langle X_1|E\rangle)$ can execute either $\tau_{\{a\}}(s_1)$ or τ. In the latter case, this initial (so non-silent) τ-transition is followed by the execution of a series of truly silent τ's in the τ-loop, until one of the process terms $\tau_{\{a\}}(s_i)$ for $i \in \{1, \ldots, n\}$ is executed.

We now present an axiom to eliminate a cluster of τ-transitions, so that only the exits of such a cluster remain. First, a precise definition is needed of a cluster and its exits.

Let E be a guarded linear recursive specification, and $I \subseteq A$. Two recursion variables X and Y in E are in the same cluster for I if and only if there exist sequences of transitions $\langle X|E\rangle \overset{b_1}{\to} \cdots \overset{b_m}{\to} \langle Y|E\rangle$ and $\langle Y|E\rangle \overset{c_1}{\to} \cdots \overset{c_n}{\to} \langle X|E\rangle$ with $b_1, \ldots, b_m, c_1, \ldots, c_n \in I \cup \{\tau\}$.

a or aX is an exit for the cluster C if and only if:

1. a or aX is a summand at the right-hand side of the recursive equation for a recursion variable in C; and
2. in the case of aX, either $a \notin I \cup \{\tau\}$ or $X \notin C$.

Table 8 presents an axiom called cluster fair abstraction rule (CFAR) for guarded linear recursive specifications. CFAR allows us to abstract away from a cluster of actions that are renamed into τ, after which only the exits of this cluster remain. In Table 8, E is a guarded linear recursive specification. Owing to the presence of the initial action τ at the left- and right-hand side of CFAR, the initial τ-transitions of $\tau_I(\langle X|E\rangle)$ can be truly silent. If the set of exits is empty, then the empty sum at the right-hand side of CFAR represents δ.

Table 8. Cluster fair abstraction rule

CFAR If in E, X is in a cluster for I with exits $\{v_1Y_1, \ldots, v_mY_m, w_1, \ldots, w_n\}$, then

$$\tau{\cdot}\tau_I(\langle X|E\rangle) \ = \ \tau{\cdot}\tau_I(v_1\langle Y_1|E\rangle + \cdots + v_m\langle Y_m|E\rangle + w_1 + \cdots + w_n)$$

The resulting axiomatisation (the axioms for ACP$_\tau$ together with RDP, RSP and CFAR) is sound and complete for ACP$_\tau$ with guarded linear recursion modulo rooted branching bisimilarity, see [8].

7 Alternating Bit Protocol

So far we have presented a standard framework ACP$_\tau$ with guarded linear recursion for the specification and manipulation of concurrent processes. Summarising, it consists of basic operators (A, $+$, $\cdot$) to define finite processes, communication operators ($\|$, $⫡$, $|$)

to express parallelism, deadlock and encapsulation (δ, ∂_H) to force atomic actions into communication, silent step and abstraction (τ, τ_I) to make internal computations invisible, and guarded linear recursion ($\langle X|E\rangle$) to capture regular processes. These constructs form a solid basis for the analysis of a wide range of systems.

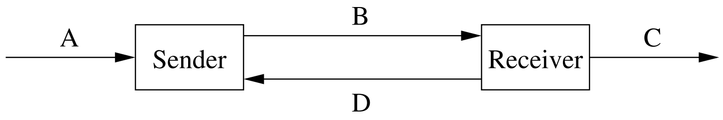
In particular, the framework is suitable for the specification and verification of network protocols. For such a verification, the desired external behaviour of the protocol is represented in the form of a process term that is in general built from the basic operators of BPA together with linear recursion. Moreover, the implementation of the protocol is represented in the form of a process term that involves the basic operators, the three parallel operators, and linear recursion. Next, the internal send and read actions of the implementation are forced into communication using an encapsulation operator, and the internal communication actions are made invisible using an abstraction operator, so that only the input/output relation of the implementation remains. If the two process terms can be equated by the axioms, then this proves that the process graphs belonging to the desired external behaviour and to the input/output relation of the implementation are rooted branching bisimilar.

An alternative to an equational correctness proof is to verify that the states in the process graph above satisfy desirable properties, expressed in some temporal logic (see, e.g., [18]). Such automated techniques to analyse process graphs are called model checking. μCRL [6,10] is a toolset for analysing process algebraic specifications in ACP combined with abstract data types; it supports equational proofs with a theorem prover, as well as generation of process graphs and model checking.

7.1 Specification of the ABP

As an example, we show how the Alternating Bit Protocol (ABP) [3] can be specified in this framework. Suppose two armies have agreed to attack a city at the same time. The two armies reside on different hills, while the city lies in between these two hills. The only way for the armies to communicate with each other is by sending messengers through the hostile city. This communication is inherently unsafe; if a messenger is caught inside the city, then the message does not reach its destination. The paradox is that in such a situation, the two armies are never able to be 100% sure that they have agreed on a time to attack the city. Namely, if one army sends the message that it will attack at say 11am, then the other army has to acknowledge reception of this message, army one has to acknowledge the reception of this acknowledgement, et cetera.

The ABP is a method to ensure successful transmission of data through a corrupted channel (such as messengers through a hostile city). This success is based on the assumption that data can be resent an unlimited number of times, and that eventually each datum will be communicated through the channel successfully. The protocol layout is depicted below.



Data elements $d_1, d_2, d_3, \ldots$ from a finite set Δ are communicated between a Sender and a Receiver. If the Sender reads a datum from channel A, then this datum is communicated through channel B to the Receiver, which sends the datum into channel C. However, channel B is corrupted, so that a message that is communicated through this channel can be turned into an error message $\perp$. Therefore, every time the Receiver receives a message via channel B, it sends an acknowledgement to the Sender via channel D, which is also corrupted.

In the ABP, the Sender attaches a bit 0 to data elements d_{2k-1} and a bit 1 to data elements d_{2k}, when they are sent into channel B. As soon as the Receiver reads a datum, it sends back the attached bit via channel D, to acknowledge reception. If the Receiver receives a corrupted message, then it sends the previous acknowledgement to the Sender once more. The Sender keeps on sending a pair (d_i, b) as long as it receives the acknowledgement $1-b$ or $\perp$. When the Sender receives the acknowledgement b, it starts sending out the next datum d_{i+1} with attached bit $1-b$, until it receives the acknowledgement $1-b$, et cetera. Alternation of the attached bit enables the Receiver to determine whether a received datum is really new, and alternation of the acknowledgement enables the Sender to determine whether it acknowledges reception of a datum or of an error message.

We give a linear recursive specification of the ABP in process algebra. First, we specify the Sender in the state that it is going to send out a datum with the bit b attached to it, represented by the recursion variable S_b for $b \in \{0,1\}$:

$$\begin{aligned}
S_b &= \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot T_{db}\\
T_{db} &= (s_{\mathrm{B}}(d,b) + s_{\mathrm{B}}(\perp))\cdot U_{db}\\
U_{db} &= r_{\mathrm{D}}(b)\cdot S_{1-b} \ + \ (r_{\mathrm{D}}(1-b) + r_{\mathrm{D}}(\perp))\cdot T_{db}
\end{aligned}$$

In state S_b, the Sender reads a datum d from channel A. Then it proceeds to state T_{db}, in which it sends datum d into channel B, with the bit b attached to it. However, the pair (d,b) may be distorted by the channel, so that it becomes the error message $\perp$. Next, the system proceeds to state U_{db}, in which it expects to receive the acknowledgement b through channel D, ensuring that the pair (d,b) has reached the Receiver unscathed. If the correct acknowledgement b is received, then the system proceeds to state S_{1-b}, in which it is going to send out a datum with the bit $1-b$ attached to it. If the acknowledgement is either the wrong bit $1-b$ or the error message $\perp$, then the system proceeds to state T_{db}, to send the pair (d,b) into channel B once more.

Next, we specify the Receiver in the state that it is expecting to receive a datum with the bit b attached to it, represented by the recursion variable R_b for $b \in \{0,1\}$:

$$\begin{aligned}
R_b &= \sum_{d'\in\Delta} \{r_{\mathrm{B}}(d',b)\cdot s_{\mathrm{C}}(d')\cdot Q_b + r_{\mathrm{B}}(d',1-b)\cdot Q_{1-b}\} \ + \ r_{\mathrm{B}}(\perp)\cdot Q_{1-b}\\
Q_b &= (s_{\mathrm{D}}(b) + s_{\mathrm{D}}(\perp))\cdot R_{1-b}
\end{aligned}$$

In state R_b there are two possibilities.

1. If in R_b the Receiver reads a pair (d',b) from channel B, then this constitutes new information, so the datum d' is sent into channel C. Then the Receiver proceeds to state Q_b, in which it sends acknowledgement b to the Sender via channel D. However, this acknowledgement may be distorted by the channel, so that it becomes

the error message $\perp$. Next, the Receiver proceeds to state R_{1-b}, in which it is expecting to receive a datum with the bit $1-b$ attached to it.

2. If in R_b the Receiver reads a pair $(d', 1-b)$ or an error message $\perp$ from channel B, then this does not constitute new information. So then the Receiver proceeds to state Q_{1-b} straight away, to send acknowledgement $1-b$ to the Sender via channel D. However, this acknowledgement may be distorted by the channel, so that it becomes the error message $\perp$. Next, the Receiver proceeds to state R_b again.

A send and a read action of the same message ((d, b), b, or $\perp$) over the same internal channel (B or D) communicate with each other:

$$\begin{array}{ll}\gamma(s_{\mathrm{B}}(d,b), r_{\mathrm{B}}(d,b)) \equiv c_{\mathrm{B}}(d,b) & \gamma(s_{\mathrm{D}}(b), r_{\mathrm{D}}(b)) \equiv c_{\mathrm{D}}(b)\\ \gamma(s_{\mathrm{B}}(\perp), r_{\mathrm{B}}(\perp)) \equiv c_{\mathrm{B}}(\perp) & \gamma(s_{\mathrm{D}}(\perp), r_{\mathrm{D}}(\perp)) \equiv c_{\mathrm{D}}(\perp)\end{array}$$

for $d \in \Delta$ and $b \in \{0,1\}$. All other communications between actions result in δ.

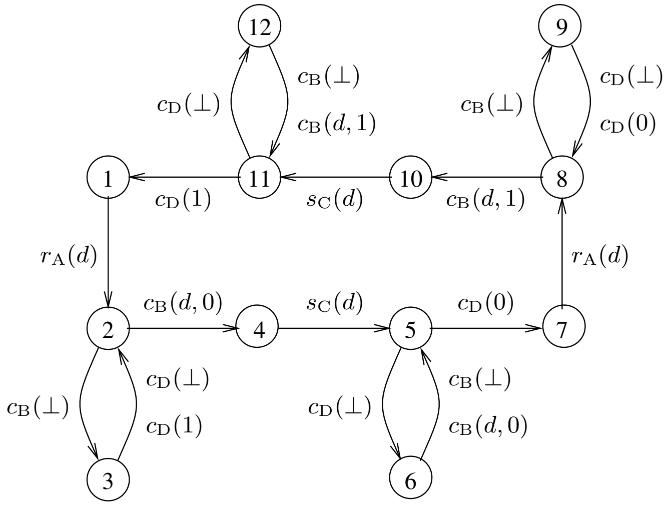
The recursive specification E of the ABP, consisting of the recursive equations for the recursion variables S_b, T_{db}, U_{db}, R_b, and Q_b for $d \in \Delta$ and $b \in \{0,1\}$, can easily be transformed into linear form by introducing extra recursion variables to represent $s_{\mathrm{C}}(d')\cdot Q_b$ for $d' \in \Delta$ and $b \in \{0,1\}$. In the remainder of this section, for notational convenience, process terms $\langle X|E\rangle$ are abbreviated to X. The desired concurrent system is obtained by putting R_0 and S_0 in parallel, encapsulating send and read actions over the internal channels B and D, and abstracting away from communication actions over these channels. That is, the ABP is expressed by the process term

$$\tau_I(\partial_H(R_0 \| S_0))$$

with

$$\begin{aligned} H &\equiv \{s_{\mathrm{B}}(d,b), r_{\mathrm{B}}(d,b), s_{\mathrm{D}}(b), r_{\mathrm{D}}(b) \mid d \in \Delta, b \in \{0,1\}\}\\ &\quad \cup \{s_{\mathrm{B}}(\perp), r_{\mathrm{B}}(\perp), s_{\mathrm{D}}(\perp), r_{\mathrm{D}}(\perp)\}\\ I &\equiv \{c_{\mathrm{B}}(d,b), c_{\mathrm{D}}(b) \mid d \in \Delta, b \in \{0,1\}\} \cup \{c_{\mathrm{B}}(\perp), c_{\mathrm{D}}(\perp)\}.\end{aligned}$$

The process graph of $\partial_H(R_0 \| S_0)$ is depicted below. Initially, in state 1, a datum d is read from channel A, resulting in state 2. Then an error message $\perp$ is communicated through channel B zero or more times, each time invoking an incorrect acknowledgement 1 or $\perp$. Finally, the pair $(d, 0)$ is communicated through channel B, resulting in state 4. Then datum d is sent into channel C, to reach state 5. The corrupted acknowledgement $\perp$ is communicated through channel D zero or more times, each time invoking a renewed attempt to communicate the pair $(d, 0)$ through channel B. Finally, acknowledgement 0 is communicated through channel D, resulting in state 7. There the same process is repeated, with the distinction that the bit 1 attached to the datum that is communicated through channel B. Note that states 2-6 and 8-12 depend on the datum d that is read from channel A.



7.2 Verification of the ABP

This section sketches an equational proof that the process algebra specification of the ABP displays the desired external behaviour; that is, the data elements that are read from channel A by the Sender are sent into channel C by the Receiver in the same order, and no data elements are lost. In other words, the process term is a solution for the guarded recursive specification

$$X \;=\; \sum_{d\in\Delta} r_A(d)\cdot s_C(d)\cdot X$$

where action $r_A(d)$ represents "read datum d from channel A", and action $s_C(d)$ represents "send datum d into channel C".

First, we derive from the axioms the six equations I-VI below, which establish the transitions between states 1-7 in the bottom half of the process graph of $\partial_H(R_0\|S_0)$.

$$\begin{aligned}
\text{I}: \partial_H(R_0\|S_0) &= \textstyle\sum_{d\in\Delta} r_A(d)\cdot\partial_H(T_{d0}\|R_0)\\
\text{II}: \partial_H(T_{d0}\|R_0) &= c_B(d,0)\cdot\partial_H(U_{d0}\|(s_C(d)Q_0)) + c_B(\perp)\cdot\partial_H(U_{d0}\|Q_1)\\
\text{III}: \partial_H(U_{d0}\|Q_1) &= (c_D(1)+c_D(\perp))\cdot\partial_H(T_{d0}\|R_0)\\
\text{IV}: \partial_H(U_{d0}\|(s_C(d)Q_0)) &= s_C(d)\cdot\partial_H(Q_0\|U_{d0})\\
\text{V}: \partial_H(Q_0\|U_{d0}) &= c_D(0)\cdot\partial_H(R_1\|S_1) + c_D(\perp)\cdot\partial_H(R_1\|T_{d0})\\
\text{VI}: \partial_H(R_1\|T_{d0}) &= (c_B(d,0)+c_B(\perp))\cdot\partial_H(Q_0\|U_{d0})
\end{aligned}$$

We start with the derivation of equation I. The process term $R_0\|S_0$ can be expanded as follows. In each step, the subterms that are reduced are underlined.

$$\begin{array}{lll}
\underline{R_0\|S_0} & \overset{\text{M1}}{=} & \underline{R_0}\mathbin{⌊\!⌊} S_0 + \underline{S_0}\mathbin{⌊\!⌊} R_0 + \underline{R_0}|\underline{S_0} \\
& \overset{\text{RDP}}{=} & \underline{(\sum_{d'\in\Delta}\{r_{\text{B}}(d',0)s_{\text{C}}(d')Q_0 + r_{\text{B}}(d',1)Q_1\} + r_{\text{B}}(\bot)Q_1)\mathbin{⌊\!⌊} S_0} \\
& & + \underline{(\sum_{d\in\Delta} r_{\text{A}}(d)T_{d0})\mathbin{⌊\!⌊} R_0} \\
\multicolumn{3}{l}{+ \underline{(\sum_{d'\in\Delta}\{r_{\text{B}}(d',0)s_{\text{C}}(d')Q_0 + r_{\text{B}}(d',1)Q_1\} + r_{\text{B}}(\bot)Q_1)|(\sum_{d\in\Delta} r_{\text{A}}(d)T_{d0})}} \\
& \overset{\text{LM4,CM9,10}}{=} & \sum_{d'\in\Delta}\{\underline{(r_{\text{B}}(d',0)s_{\text{C}}(d')Q_0)\mathbin{⌊\!⌊} S_0} + \underline{(r_{\text{B}}(d',1)Q_1)\mathbin{⌊\!⌊} S_0}\} \\
& & + \underline{(r_{\text{B}}(\bot)Q_1)\mathbin{⌊\!⌊} S_0} + \sum_{d\in\Delta}\underline{(r_{\text{A}}(d)T_{d0})\mathbin{⌊\!⌊} R_0} \\
\multicolumn{3}{l}{+ \sum_{d'\in\Delta}\sum_{d\in\Delta}\{\underline{(r_{\text{B}}(d',0)s_{\text{C}}(d')Q_0)|(r_{\text{A}}(d)T_{d0})} + \underline{(r_{\text{B}}(d',1)Q_1)|(r_{\text{A}}(d)T_{d0})}\}} \\
& & + \sum_{d\in\Delta}\underline{(r_{\text{B}}(\bot)Q_1)|(r_{\text{A}}(d)T_{d0})} \\
& \overset{\text{LM3,CM8}}{=} & \sum_{d'\in\Delta}\{r_{\text{B}}(d',0)((s_{\text{C}}(d')Q_0)\|S_0) + r_{\text{B}}(d',1)(Q_1\|S_0)\} \\
& & + r_{\text{B}}(\bot)(Q_1\|S_0) + \sum_{d\in\Delta} r_{\text{A}}(d)(T_{d0}\|R_0) \\
& & + \sum_{d'\in\Delta}\sum_{d\in\Delta}\{\underline{\delta((s_{\text{C}}(d')Q_0)\|T_{d0})} + \underline{\delta(Q_1\|T_{d0})}\} \\
& & + \sum_{d\in\Delta}\underline{\delta(Q_1\|T_{d0})} \\
& \overset{\text{A6,7}}{=} & \sum_{d'\in\Delta}\{r_{\text{B}}(d',0)((s_{\text{C}}(d')Q_0)\|S_0) + r_{\text{B}}(d',1)(Q_1\|S_0)\} \\
& & + r_{\text{B}}(\bot)(Q_1\|S_0) + \sum_{d\in\Delta} r_{\text{A}}(d)(T_{d0}\|R_0).
\end{array}$$

Next, we expand the process term $\partial_H(R_0\|S_0)$.

$$\begin{array}{lll}
\partial_H(\underline{R_0\|S_0}) & = & \underline{\partial_H(\sum_{d'\in\Delta}\{r_{\text{B}}(d',0)((s_{\text{C}}(d')Q_0)\|S_0) + r_{\text{B}}(d',1)(Q_1\|S_0)\}} \\
& & \underline{+ r_{\text{B}}(\bot)(Q_1\|S_0) + \sum_{d\in\Delta} r_{\text{A}}(d)(T_{d0}\|R_0))} \\
& \overset{\text{D4}}{=} & \sum_{d'\in\Delta}\{\underline{\partial_H(r_{\text{B}}(d',0)((s_{\text{C}}(d')Q_0)\|S_0))} + \underline{\partial_H(r_{\text{B}}(d',1)(Q_1\|S_0))}\} \\
& & + \underline{\partial_H(r_{\text{B}}(\bot)(Q_1\|S_0))} + \sum_{d\in\Delta}\underline{\partial_H(r_{\text{A}}(d)(T_{d0}\|R_0))} \\
& \overset{\text{D1,2,5}}{=} & \sum_{d'\in\Delta}\{\underline{\delta\partial_H((s_{\text{C}}(d')Q_0)\|S_0)} + \underline{\delta\partial_H(Q_1\|S_0)}\} + \underline{\delta\partial_H(Q_1\|S_0)} \\
& & + \sum_{d\in\Delta} r_{\text{A}}(d)\partial_H(T_{d0}\|R_0) \\
& \overset{\text{A6,7}}{=} & \sum_{d\in\Delta} r_{\text{A}}(d)\partial_H(T_{d0}\|R_0).
\end{array}$$

This completes the proof of equation I. Similar to equation I, we can derive the remaining equations II-VI. These derivations are sketched below.

$$\begin{array}{l}
T_{d0}\|R_0 = (s_{\text{B}}(d,0) + s_{\text{B}}(\bot))(U_{d0}\|R_0) \\
\qquad + \sum_{d'\in\Delta}\{r_{\text{B}}(d',0)((s_{\text{C}}(d')Q_0)\|T_{d0}) + r_{\text{B}}(d',1)(Q_1\|T_{d0})\} \\
\qquad + r_{\text{B}}(\bot)(Q_1\|T_{d0}) + c_{\text{B}}(d,0)(U_{d0}\|(s_{\text{C}}(d)Q_0)) + c_{\text{B}}(\bot)(U_{d0}\|Q_1) \\
\partial_H(T_{d0}\|R_0) = c_{\text{B}}(d,0)\partial_H(U_{d0}\|(s_{\text{C}}(d)Q_0)) + c_{\text{B}}(\bot)\partial_H(U_{d0}\|Q_1)
\end{array}$$

$$\begin{array}{l}
U_{d0}\|Q_1 = r_{\text{D}}(0)(S_1\|Q_1) + (r_{\text{D}}(1) + r_{\text{D}}(\bot))(T_{d0}\|Q_1) \\
\qquad + (s_{\text{D}}(1) + s_{\text{D}}(\bot))(R_0\|U_{d0}) + (c_{\text{D}}(1) + c_{\text{D}}(\bot))(T_{d0}\|R_0) \\
\partial_H(U_{d0}\|Q_1) = (c_{\text{D}}(1) + c_{\text{D}}(\bot))\partial_H(T_{d0}\|R_0)
\end{array}$$

$$\begin{aligned}U_{d0} \| (s_C(d)Q_0) &= r_D(0)(S_1 \| (s_C(d)Q_0)) + (r_D(1) + r_D(\perp))(T_{d0} \| (s_C(d)Q_0)) \\ &\quad + s_C(d)(Q_0 \| U_{d0}) \\ \partial_H(U_{d0} \| (s_C(d)Q_0)) &= s_C(d)\partial_H(Q_0 \| U_{d0})\end{aligned}$$

$$\begin{aligned}Q_0 \| U_{d0} &= (s_D(0) + s_D(\perp))(R_1 \| U_{d0}) + r_D(0)(S_1 \| Q_0) \\ &\quad + (r_D(1) + r_D(\perp))(T_{d0} \| Q_0) + c_D(0)(R_1 \| S_1) + c_D(\perp)(R_1 \| T_{d0}) \\ \partial_H(Q_0 \| U_{d0}) &= c_D(0)\partial_H(R_1 \| S_1) + c_D(\perp)\partial_H(R_1 \| T_{d0})\end{aligned}$$

$$\begin{aligned}R_1 \| T_{d0} &= \sum_{d' \in \Delta} \{r_B(d',1)((s_C(d')Q_1) \| T_{d0}) + r_B(d',0)(Q_0 \| T_{d0})\} \\ &\quad + r_B(\perp)(Q_0 \| T_{d0}) + (s_B(d,0) + s_B(\perp))(U_{d0} \| R_1) \\ &\quad + (c_B(d,0) + c_B(\perp))(Q_0 \| U_{d0}) \\ \partial_H(R_1 \| T_{d0}) &= (c_B(d,0) + c_B(\perp))\partial_H(Q_0 \| U_{d0})\end{aligned}$$

Note that the process term $\partial_H(R_1 \| S_1)$ in the right-hand side of equation V is not the left-hand side of an equation I-VI. We proceed to expand $\partial_H(R_1 \| S_1)$. That is, similar to equations I-VI, the following six equations VII-XII can be derived, which establish the transitions between states 7-12 and 1 in the top half of the process graph of $\partial_H(R_0 \| S_0)$. The derivations of these equations are left to the reader.

$$\begin{aligned}\text{VII}: \partial_H(R_1 \| S_1) &= \textstyle\sum_{d \in \Delta} r_A(d) \cdot \partial_H(T_{d1} \| R_1) \\ \text{VIII}: \partial_H(T_{d1} \| R_1) &= c_B(d,1) \cdot \partial_H(U_{d1} \| (s_C(d)Q_1)) + c_B(\perp) \cdot \partial_H(U_{d1} \| Q_0) \\ \text{IX}: \partial_H(U_{d1} \| Q_0) &= (c_D(0) + c_D(\perp)) \cdot \partial_H(T_{d1} \| R_1) \\ \text{X}: \partial_H(U_{d1} \| (s_C(d)Q_1)) &= s_C(d) \cdot \partial_H(Q_1 \| U_{d1}) \\ \text{XI}: \partial_H(Q_1 \| U_{d1}) &= c_D(1) \cdot \partial_H(R_0 \| S_0) + c_D(\perp) \cdot \partial_H(R_0 \| T_{d1}) \\ \text{XII}: \partial_H(R_0 \| T_{d1}) &= (c_B(d,1) + c_B(\perp)) \cdot \partial_H(Q_1 \| U_{d1})\end{aligned}$$

Thus, we can derive algebraically the relations depicted in the process graph of $\partial_H(R_0 \| S_0)$. Owing to equations I-XII, RSP yields

$$\partial_H(R_0 \| S_0) = \langle X_1 | E \rangle \tag{1}$$

where E denotes the linear recursive specification

$$\begin{array}{ll}\{\; X_1 = \sum_{d' \in \Delta} r_A(d') \cdot X_{2d'}, & Y_1 = \sum_{d' \in \Delta} r_A(d') \cdot Y_{2d'}, \\ \;\; X_{2d} = c_B(d,0) \cdot X_{4d} + c_B(\perp) \cdot X_{3d}, & Y_{2d} = c_B(d,1) \cdot Y_{4d} + c_B(\perp) \cdot Y_{3d}, \\ \;\; X_{3d} = (c_D(1) + c_D(\perp)) \cdot X_{2d}, & Y_{3d} = (c_D(0) + c_D(\perp)) \cdot Y_{2d}, \\ \;\; X_{4d} = s_C(d) \cdot X_{5d}, & Y_{4d} = s_C(d) \cdot Y_{5d}, \\ \;\; X_{5d} = c_D(0) \cdot Y_1 + c_D(\perp) \cdot X_{6d}, & Y_{5d} = c_D(1) \cdot X_1 + c_D(\perp) \cdot Y_{6d}, \\ \;\; X_{6d} = (c_B(d,0) + c_B(\perp)) \cdot X_{5d}, & Y_{6d} = (c_B(d,1) + c_B(\perp)) \cdot Y_{5d} \\ \;\; \mid d \in \Delta \;\}. & \end{array}$$

We proceed to prove that the process term $\tau_I(\langle X_1 | E \rangle)$ exhibits the desired external behaviour of the ABP. After application of the abstraction operator τ_I to the process term $\langle X_1|E\rangle$, the loops of communication actions in the process graph of $\partial_H(R_0\|S_0)$ (between states 2-3, states 5-6, states 8-9, and states 11-12) become τ-loops. These loops can be removed using CFAR. For example, for $d \in \Delta$ the recursion variables X_{2d} and X_{3d} form a cluster for I with exit $c_{\mathrm{B}}(d,0)\cdot X_{4d}$, so

$$\begin{aligned} r_{\mathrm{A}}(d)\cdot\tau_I(\langle X_{2d}|E\rangle) &\overset{\text{CFAR}}{=} r_{\mathrm{A}}(d)\cdot\tau_I(c_{\mathrm{B}}(d,0)\,\langle X_{4d}|E\rangle) \\ &\overset{\text{TI2,5,B1}}{=} r_{\mathrm{A}}(d)\cdot\tau_I(\langle X_{4d}|E\rangle). \end{aligned} \tag{2}$$

Similarly, CFAR together with TI2,5 and B1 can be applied to eliminate the other three loops of communication actions. Thus, we derive the following equations:

$$s_{\mathrm{C}}(d)\cdot\tau_I(\langle X_{5d}|E\rangle) = s_{\mathrm{C}}(d)\cdot\tau_I(\langle Y_1|E\rangle) \tag{3}$$

$$r_{\mathrm{A}}(d)\cdot\tau_I(\langle Y_{2d}|E\rangle) = r_{\mathrm{A}}(d)\cdot\tau_I(\langle Y_{4d}|E\rangle) \tag{4}$$

$$s_{\mathrm{C}}(d)\cdot\tau_I(\langle Y_{5d}|E\rangle) = s_{\mathrm{C}}(d)\cdot\tau_I(\langle X_1|E\rangle). \tag{5}$$

Applying RDP, TI1,4,5, and equations (2) and (3) we derive

$$\begin{aligned} \tau_I(\langle X_1|E\rangle) &\overset{\text{RDP,TI1,4,5}}{=} \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot\tau_I(\langle X_{2d}|E\rangle) \\ &\overset{(2)}{=} \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot\tau_I(\langle X_{4d}|E\rangle) \\ &\overset{\text{RDP,TI1,5}}{=} \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot s_{\mathrm{C}}(d)\cdot\tau_I(\langle X_{5d}|E\rangle) \\ &\overset{(3)}{=} \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot s_{\mathrm{C}}(d)\cdot\tau_I(\langle Y_1|E\rangle). \end{aligned} \tag{6}$$

Likewise, applying RDP, TI1,4,5, and equations (4) and (5) we can derive

$$\tau_I(\langle Y_1|E\rangle) = \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot s_{\mathrm{C}}(d)\cdot\tau_I(\langle X_1|E\rangle). \tag{7}$$

Equations (6) and (7) together with RSP enable us to derive the following equation:

$$\tau_I(\langle X_1|E\rangle) = \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot s_{\mathrm{C}}(d)\cdot\tau_I(\langle X_1|E\rangle).$$

In combination with equation (1) this yields

$$\tau_I(\partial_H(R_0\|S_0)) = \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot s_{\mathrm{C}}(d)\cdot\tau_I(\partial_H(R_0\|S_0)).$$

In other words, the ABP exhibits the desired external behaviour. This finishes the verification of the ABP.

References

1. Aceto, L., Fokkink, W.J., Verhoef, C.: Structural operational semantics. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 197–292. Elsevier, Amsterdam (2001)
2. Baeten, J.C.M.: A brief history of process algebra. Theoretical Computer Science 335(2-3), 131–146 (2005)
3. Bartlett, K.A., Scantlebury, R.A., Wilkinson, P.T.: A note on reliable full-duplex transmission over half-duplex links. Communications of the ACM 12(5), 260–261 (1969)
4. Basten, T.: Branching bisimilarity is an equivalence indeed! Information Processing Letters 58(3), 141–147 (1996)
5. Bergstra, J., Heering, J., Klint, P.: Module algebra. Journal of the ACM 37(2), 335–372 (1990)
6. Blom, S., Fokkink, W.J., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.: μCRL: A toolset for analysing algebraic specification. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 250–254. Springer, Heidelberg (2001)
7. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. Journal of the ACM 42(2), 458–487 (1995)
8. Fokkink, W.J.: Introduction to Process Algebra. Springer, Heidelberg (2000)
9. Fokkink, W.J.: Rooted branching bisimulation as a congruence. Journal of Computer and System Sciences 60(1), 13–37 (2000)
10. Fokkink, W.J.: Modelling Distributed Systems. Springer, Heidelberg (2007)
11. Fokkink, W.J., Verhoef, C.: A conservative look at operational semantics with variable binding. Information and Computation 146(1), 24–54 (1998)
12. van Glabbeek, R.J.: A complete axiomatization for branching bisimulation congruence of finite-state behaviours. In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 473–484. Springer, Heidelberg (1993)
13. van Glabbeek, R.J.: What is branching time and why to use it? In: Nielsen, M. (ed.) The Concurrency Column. Bulletin of the EATCS, vol. 53, pp. 190–198 (1994)
14. Groote, J.F.: Process Algebra and Structured Operational Semantics. PhD thesis, University of Amsterdam (1991)
15. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
16. Moller, F.: The importance of the left merge operator in process algebras. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 752–764. Springer, Heidelberg (1990)
17. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
18. Stirling, C.: Modal and Temporal Properties of Processes. Springer, Heidelberg (2001)
19. Tanenbaum, A.S.: Computer Networks. Prentice-Hall, Englewood Cliffs (1981)
20. Terese: Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
21. Verhoef, C.: A congruence theorem for structured operational semantics with predicates and negative premises. Nordic Journal of Computing 2(2), 274–302 (1995)